Webalkalmazások fejlesztése

Microsoft Azure

2016.

# Bevezetés

## Célkitűzés

A labor célja, hogy a hallgatók megismerkedjenek az Azure Web Appok létrehozásának menetével. A feladatok során egyszerű web appokat készítünk el ASP.NET és NodeJS keretrendszerekben, ezeket publikáljuk a felhőbe. Kipróbáljuk a Web jobokat és a continuous integration-t is.

A labor nem igényel erős webfejlesztési ismereteket sem ASP.NET-ben, sem NodeJS-ben, viszont feltételezi, hogy a hallgató tisztában van az Azure Storage fogalmaival és használatával.

## Előfeltételek

A labor elvégzéséhez szükséges eszközök:

* Visual Studio 2015
* Visual Studio Code (Gittel telepítve; a NodeJS feladathoz)
* Azure előfizetés
  + Jogosultság Web app-ok létrehozására
  + Tárhelyfiók kapcsolódási adatok
* A KuduScript feladathoz az Azure Cross-Platform command line interface-re is szükségünk van:  
  <https://azure.microsoft.com/hu-hu/documentation/articles/xplat-cli-install/>

## Amit érdemes átnézned

* Adattárolás 1 előadás/labor: blob storage
* Kapcsolódó előadás

## Oktatói útmutató

A WebJob feladat megoldásához tárhelyfiókra van szükség. A hallgatók Imagine (not billable) subscription-je nem alkalmas tárhelyfiók létrehozására, így ők az oktató által létrehozott tárhelyfiókot használhatják. Storage Emulator itt nem használható, mivel a felhőben futó WebJob nem tud az emulátorunkhoz csatlakozni.

A teljes labor csak nagyon erőltetett tempóban fér bele 90 percbe, ezért a következő megközelítéseket javasoljuk:

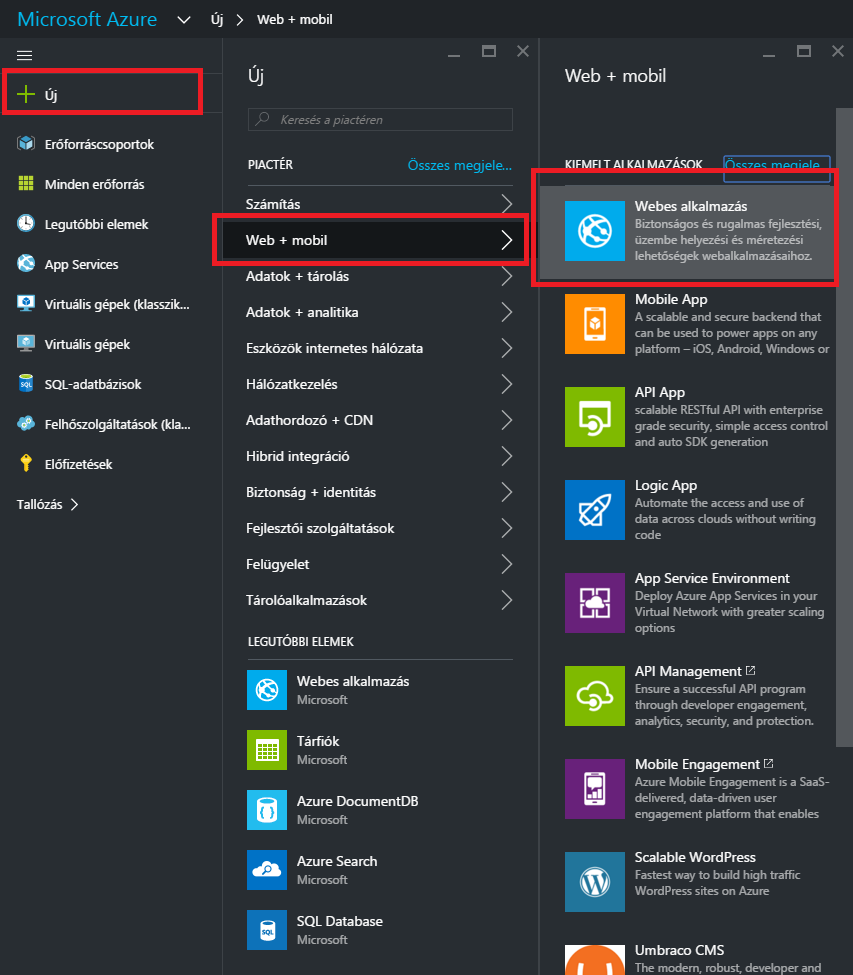
* Az anyag 4, független feladatból áll. Válogassunk belőlük saját preferenciánk szerint!
* Mivel a labornak nem célja sem a webfejlesztés, sem az Azure Storage bemutatása, ezért ezeket a kódrészleteket előre kitehetjük valahova, ahonnan a hallgatók a labor során lemásolhatják (a webjob résszel kapcsolatos feladatokat viszont érdemes végigcsinálni!).

# ASP.NET Web app

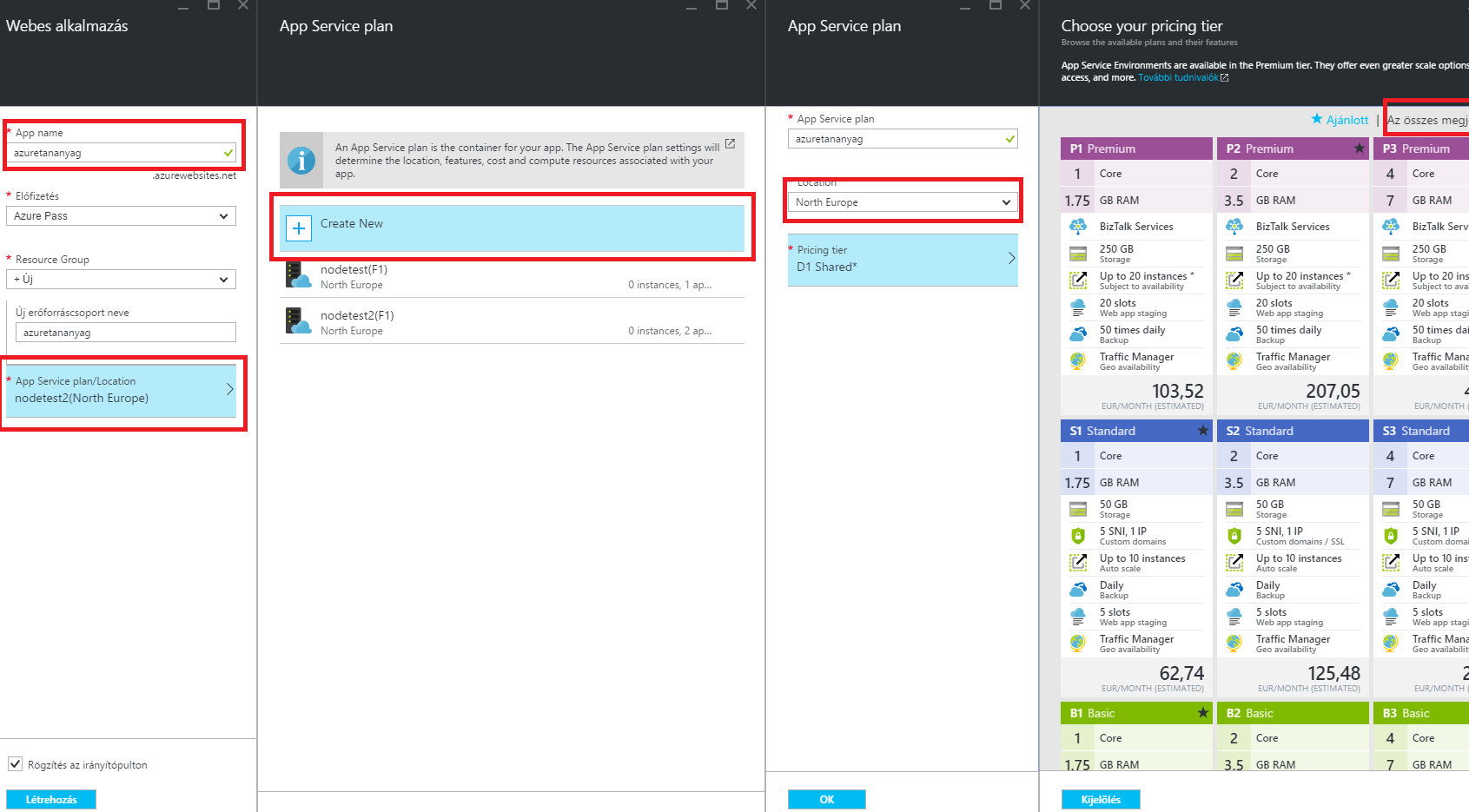
Ebben a feladatban egy egyszerű ASP.NET webalkalmazást fogunk elkészíteni és publikálni web appként a felhőbe. A feladat fő fókusza nem a webfejlesztés, hanem a publikálás menete. Majd egy kis módosítás után a web apphoz egy egyszerű web jobot csatolunk. Végül a népszerű NodeJS platformon készült alkalmazások publikálási lehetőségeit is megnézzük, kitekintésként pedig megnézzük, hogyan tudjuk a telepítési folyamatot is teljesen testre szabhatóvá tenni.

## Web app szolgáltatás létrehozása

Első lépésként létre kell hoznunk egy Web Appot, ahová publikálni tudjuk majd a webalkalmazásunkat.

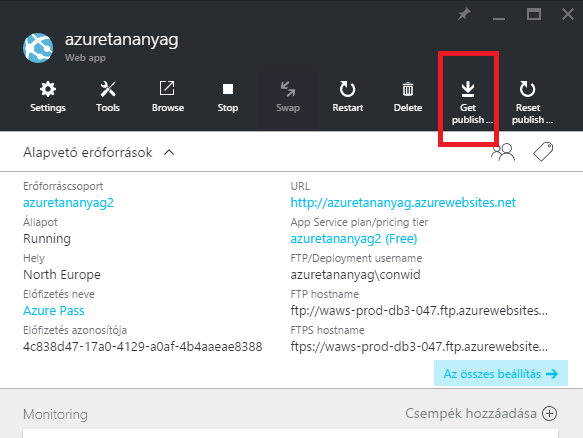


Az ezután felbukkanó ablakban tudjuk a létrejövő webalkalmazás néhány tulajdonságát beállítani:



Az App Name tetszőleges, de egyedi kell, hogy legyen. A Resource group szintén lehet most tetszőleges, ha van már létrehozva, akkor azt is használhatjuk. Az App Service plan esetében hozzunk létre újat. A név itt is tetszőleges lehet, a Location legyen North Europe. A Pricing Tier kiválasztásakor kattintsunk a jobb felső sarokban „Az összes megjelenítése” gombra, ekkor minden opció elérhetővé válik; válasszuk az F1 Free opciót. Ne felejtsük el minden ablakban megnyomni a „Kijelölés”/”OK”/”Létrehozás” stb. gombokat.

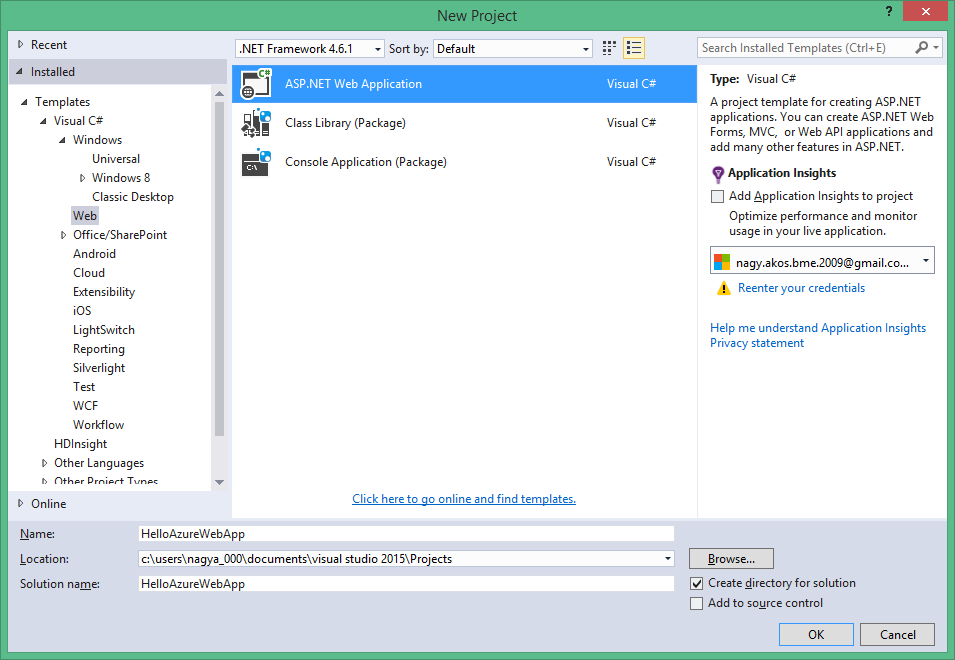
Ha elkészült a webalkalmazás, töltsük le a Publish profile-t:



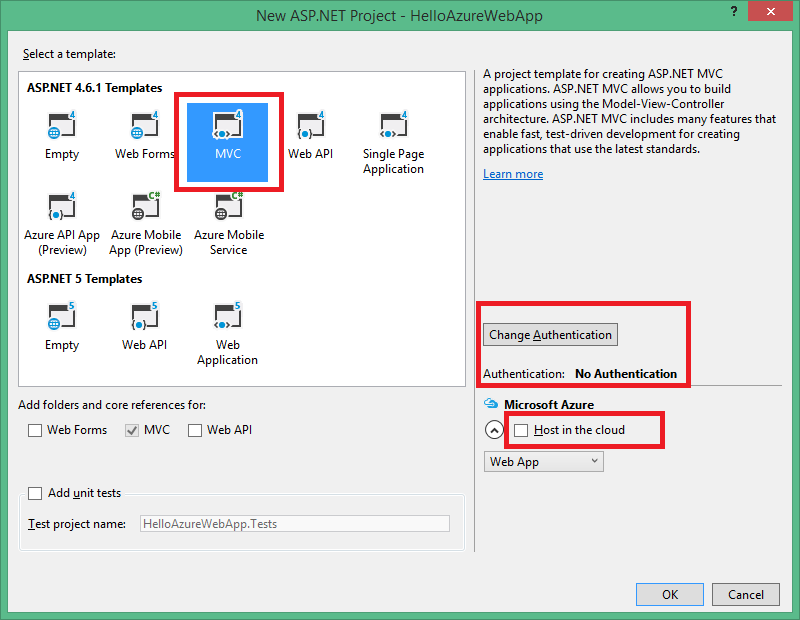
## Webalkalmazás létrehozása Visual Studioban

Hozzunk létre egy új webalkalmazást Visual Studioban!

1. File / New Project / Web / ASP.NET Web Application

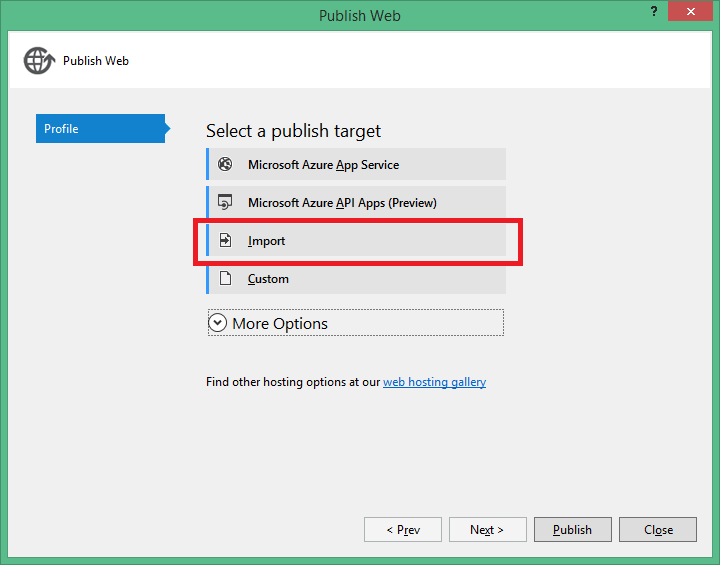


1. A következő varázsló-ablakban állítsuk be a következőket:



ASP.NET 4.6.1 MVC alkalmazás. A Change Authentication gombra kattintva válasszuk ki a „No authentication” opciót és egyelőre a „Host in the cloud” checkbox se legyen bepipálva! Ezután kattintsunk az OK gombra.

1. A generálódó üres sablonalkalmazást akár el is indíthatjuk (F5). Ha sikerült lokálisan elindítanunk, akkor telepítsük ki Azure-be.
2. Kattintsunk jobb gombbal a projektre, majd válasszuk a Publish opciót!
3. Az előugró varázslóablakban válasszuk az Import lehetőséget:



A felugró ablakban tallózzuk ki az előző fejezet végén lementett Publish profile fájlt!

1. Végignézhetjük a varázsló többi ablakát is. A connection alatt a web apphoz történő kapcsolódás fontosabb elemeit láthatjuk, a Settings alatt néhány telepítési beállítás van (pl.: milyen Build konfigurációt használjunk), a preview ablakban pedig azt nézhetjük meg, hogy a legutóbbi változásokhoz képest milyen új build outputok kerülnek majd telepítésre.

Érdemes felhívni a figyelmet arra, hogy mindig Release (vagy Release alapján létrehozott) Build konfigurációban publikálunk!

1. Ha végeztünk az opciók átnézésével, nyomjuk meg a Publish gombot! Ha mindent jól csináltunk, egy kis gondolkodás után az alkalmazásunk kitelepül és böngészőben megnyílik a webalkalmazásunk.

A labor során a lehető legkevesebb webfejlesztési gyakorlat kerül bemutatásra. A cél az, hogy a felhős fejlesztés speciális lépéseivel megismerkedjenek a hallgatók. Itt mondjuk el, hogy tetszőleges ASP.NET webalkalmazást ki tudunk így telepíteni, csak rajtunk áll, hogy pontosan mit kódolunk bele.

1. Ha időnk engedi, érdemes kicsit nézelődni a beállítások között (General/Application Settings; itt például láthatjuk, hogy milyen támogatott platformok vannak vagy hogyan tudunk felvenni környezeti változókat, megadni webszerver-specifikus beállításokat).

# Webjob

Adott egy képfeltöltő form, amelyen keresztül egy blobba tudunk képeket feltölteni. Ha egy új kép kerül a blobba, akkor szeretnék a képekből egy thumbnail-t készíteni és ezt is beszúrni a blobba. Erre az egyszerű feladatra készítünk el egy webjobot.

## Webjob alkalmazás készítése

A webjobokat klasszikus konzol alkalmazásként kell, hogy megírjuk. Ebben megfelelő metódushívásokkal és attribútumokkal tudjuk jelezni, hogy itt egy webjobról van szó, amelyet aztán később majd a portálon feltölthetünk és használhatunk.

1. Hozzunk létre egy Azure WebJob alkalmazást Visual Studio-ban a Webalkalmazás projektje mellé (File / New Project / Cloud / Azure Web Job)! Ez lényegében egy egyszerű konzolos alkalmazás, de telepítve vannak bele a megfelelő NuGet csomagok és a config fájl is tartalmaz néhány kiinduló értéket. Első lépésként fordítsuk le a projektet, hogy a Nuget csomagok helyreálljanak.
2. A Program.cs állomány Main() metódusába a következő két sort találjuk (+ kommentek):

JobHost host = new JobHost();

host.RunAndBlock();

Ezzel a két sorral létrehozunk egy JobHost objektumot, majd ennek RunAndBlock() metódusával az aktuális szálon elkezdjük futtatni az összes ütemezett feladatot. Ütemezett feladatunk még nincs, ezt a következő lépésben hozzuk létre.

1. A Function.cs szolgál arra, hogy megvalósítsuk az ütemezett feladatunkat. Nézzük meg a benne lévő példakódot (ha a „queue” nevű sorba bekerül egy üzenet, akkor ezt kinaplózza). Utaljunk az előadásra, ahol szerepel az, hogy különböző attribútumokkal kell ellátni a futtatandó kódot, és ez „automatikusan” meghívódik (ezt csinálja a RunAndBlock() metódus). Töröljük az osztály tartalmát és vegyük fel azt a metódust, amely a mi funkciónkat megvalósítja:

public static void Resize(

[BlobTrigger("images/{name}")] WebImage input,

[Blob("images-resized/{name}")] out WebImage output)

{

var width = 180;

var height = Convert.ToInt32(input.Height \* 180 / input.Width);

output = input.Resize(width, height);

}

A WebImage osztály használatához a System.Web.Helpers referenciát fel kell vennünk a projektbe!

Amennyiben a Visual Studio nem találja a 'System.Web.Helpers' referenciát, hozzá kell adni a 'Microsoft.AspNet.WebHelpers' NuGet csomagot a projekthez!

Magyarázat:

* A BlobTrigger attribútummal azt adjuk meg, hogy egyrészt ennek a feladatnak akkor kell lefutnia, amikor egy új blobot töltenek fel az images containerbe, másrészt pedig azt, hogy ennek a feltöltött blobnak be kell kerülnie az input paraméterbe. A kimeneti paramétert Blob attribútummal látjuk el, ezzel jelezve, hogy ezt az eredményt a megadott konténerbe a megadott névvel bele kell másolni.
* Mind a BlobTrigger, mind a Blob attribútum konstruktora paraméterként egy sablonstringet vesz át, amiben megadhatjuk, hogy milyen blobokra kell, hogy lefusson a job. Itt most az images konténerben lévő blobkra fut a művelet (ez a sablonnak statikus része), a {} jelek között lévő {name} sablonrészt pedig arra használjuk, hogy a Blob attribútumban is ezt használjuk. Így a kimeneti blob neve ugyanaz lesz, mint a bemeneti blob neve, de egy másik konténerbe kerül majd át (más a Blob attribútum sablonjának statikus része).

1. A BlobTriggert és Blob attribútumokat olyan típusú paraméterekre tehetjük rá, amelyek a Stream típussal kompatiblisek. A WebImage nem ilyen. Használhatnánk helyette Stream-et, de a WebImage extra szolgáltatásai (Height, Width) hasznosak számunkra. Az ilyen esetekre felvehetünk egy custom binder-t, amely tud a Stream és a WebImage között konvertálni. Vegyünk fel egy új osztályt a projektbe (WebImageBinder.cs) az alábbi kóddal:

public class WebImageBinder : ICloudBlobStreamBinder<WebImage>

{

public Task<WebImage> ReadFromStreamAsync(Stream input,

System.Threading.CancellationToken cancellationToken)

{

return Task.FromResult<WebImage>(new WebImage(input));

}

public Task WriteToStreamAsync(WebImage value, Stream output,

System.Threading.CancellationToken cancellationToken)

{

var bytes = value.GetBytes();

return output.WriteAsync(bytes, 0, bytes.Length, cancellationToken);

}

}

1. Ezzel elkészült a webjobunk. Az utolsó dolog, hogy meg kell adni a tárhelyfiók (lásd oktatói útmutató) connection stringjét az App.configban:

<connectionStrings>

<add name="AzureWebJobsDashboard" connectionString="DefaultEndpointsProtocol=https;AccountName=[accountname];AccountKey=[accesskey]"/>

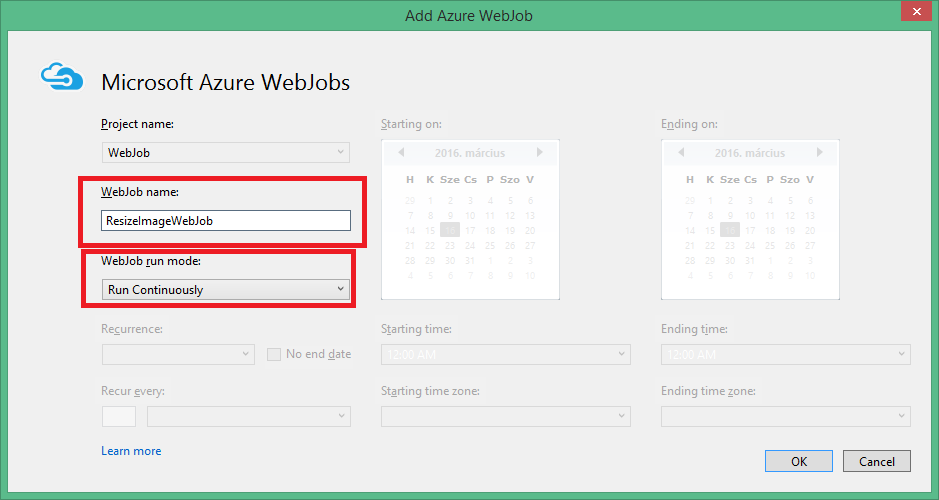
<add name="AzureWebJobsStorage" connectionString="DefaultEndpointsProtocol=https;AccountName=[accountname];AccountKey=[accesskey]"/>

</connectionStrings>

Magyarázat:

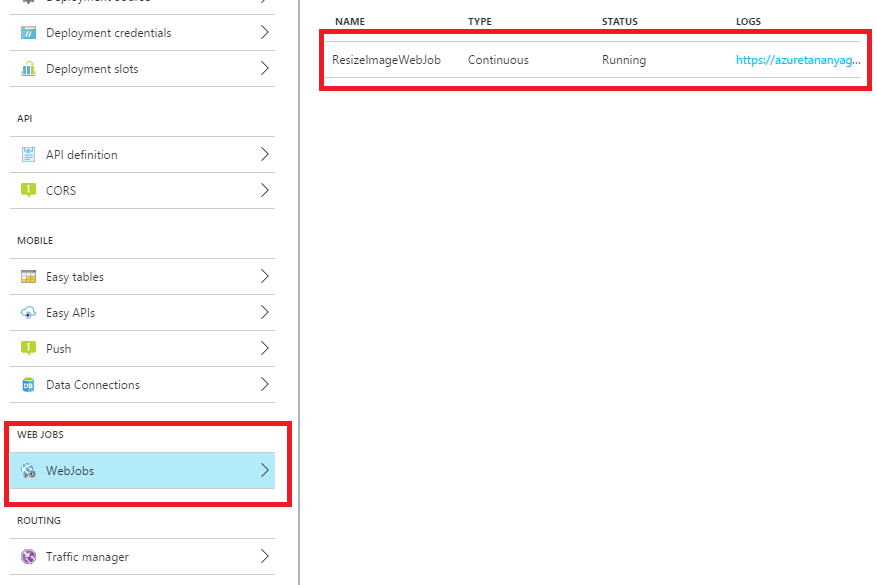
* a WebJob SDK jobjai néhány default értékkel keresik az App.configban a kulcsokat. Ezek közül az egyik az AzureWebJobsStorage, ahonnan az inputot várják és ahová az outputot teszik. A másik az AzureWebJobsDashboard, ahová naplózáshoz szükséges információk kerülnek. Saját kulccsal is felvehetjük ezeket az értékeket, ekkor ezt a JobHost osztálynak meg kell adni.

1. Ezután már publikálnunk kell a WebJobunkat. Ehhez kattintsunk jobb gombbal a projektre, majd válasszuk a Publish as Azure WebJob opciót:



A név lehet tetszőleges, a WebJob run mode legyen „Run Continuously”. Miután az OK gombra kattintunk, a webalkalmazás publikálásához hasonló varázsló fogad minket. Importáljuk ugyanazt a Publishsettings fájlt, mint korábban, majd kattintsunk a Publish gombra.

1. Ha a publikálás végigment, az Azure Portal mutatja a telepített webjobot a webalkalmazás mellett:



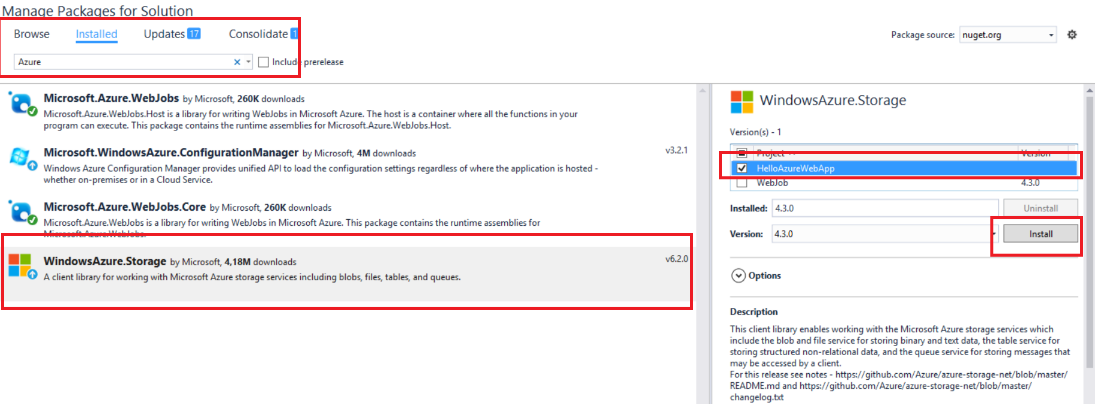
Megnézhetjük pl. a naplóbejegyzéseket a jobb oldali linkre kattintva.

Ezen a ponton dönthetünk, hogy megvalósítjuk a képfeltöltő webalkalmazást vagy sem. Ha nem szeretnénk, akkor kézzel is létrehozhatjuk az „images” konténert a blob accountban, aztán Visual Studioból feltölthetünk egy képet.

## Webalkalmazás módosítása

Módosítsuk a webalkalmazást úgy, hogy ezen keresztül történjen a feltöltés.

1. Amikor a WebJob projektet létrehoztuk, akkor ez automatikusan betöltötte a szükséges NuGet csomagokat. Ezek közül az egyik a WindowsAzure.Storage, amelyet a feltöltéshez is használnunk kell. Először tehát a Solutionön nyomjunk jobb gombot, majd a Manage NuGet Packages for Solution opciót választva telepítsük fel a WebJob projektbe már telepített NuGet csomagot a webes alkalmazásba is.



1. Ezután módosítsuk a Views/Home/Index.cshtml fájlt az alábbiaknak megfelelően (mivel a gyakorlatnak nem célja a webfejlesztés bemutatása, így ezt akár előre ki is rakhatjuk a hallgatóknak valahová, ahonnan letölthetik és beilleszthetik; új kódrészlet sárgával kiemelve):

<div class="jumbotron">

<h1>ASP.NET</h1>

<p class="lead">ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.</p>

<p><a href="http://asp.net" class="btn btn-primary btn-lg">Learn more &raquo;</a></p>

</div>

<div class="row">

<div class="col-md-4">

@using( Html.BeginForm( "Upload", "Home", FormMethod.Post, new { enctype = "multipart/form-data" } ) )

{

<input type="file" name="file" />

<input type="submit" name="Submit" id="Submit" value="Upload" />

}

</div>

</div>

<div class="row">

<div class="col-md-4">

<h2>Getting started</h2>

<p>

ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that

1. Ezután pedig a Controllers/HomeController.cs állományban a HomeController osztályt egészítsük ki az alábbi kóddal (ne felejtsük el megadni a helyes connection stringet, amely a blob accountra mutat; részletes magyarázat az alábbi kódrészletről az Adattárolás 1 gyakorlati útmutatóban található):

private static readonly string connectionString = "";

private static CloudStorageAccount account = CloudStorageAccount.Parse( connectionString );

[HttpPost]

public ActionResult Upload( HttpPostedFileBase file )

{

var client = account.CreateCloudBlobClient();

var container = client.GetContainerReference( "images" );

container.CreateIfNotExists();

container.SetPermissions( new BlobContainerPermissions {

PublicAccess = BlobContainerPublicAccessType.Off } );

var blob = container.GetBlockBlobReference( file.FileName );

var bytes = new byte[file.InputStream.Length];

file.InputStream.Read( bytes, 0, (int)file.InputStream.Length );

blob.UploadFromByteArray( bytes , 0, bytes.Length );

return RedirectToAction( "Index" );

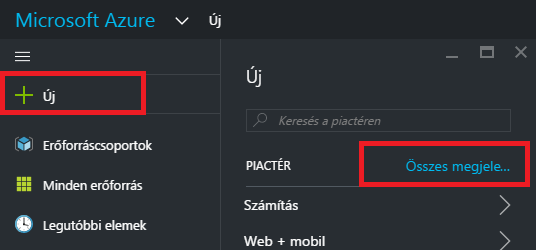
}

1. Publikáljunk újra, majd próbáljuk ki. Töltsünk fel egy fájlt, aztán Visual Studioból ellenőrizzük, hogy megtörtént-e a feltöltés. Kis idő múlva a kicsinyített másolatnak is meg kell jelennie a másik konténerben.

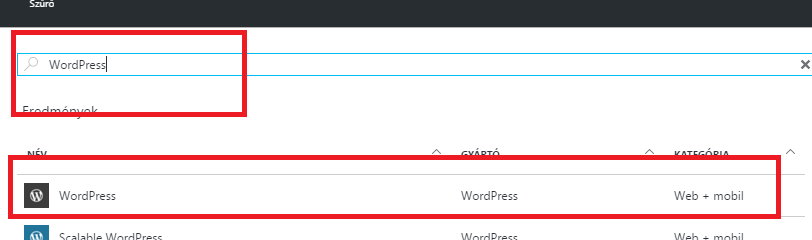
A BlobTriggereknek van két nagyon fontos tulajdonságuk. Az egyik, hogy minden blobra pontosan egyszer futnak le. A másik, hogy a belső pollozó algoritmus, amely kiváltja a triggert, a dokumentáció szerint 10 percenként fut. Így tehát akár 10 percet is várni kell, amíg megjelenik az új blob (érdemes lehet tovább folytatni addig a gyakorlatot). Ez alól kivételt képeznek azok a blobok, amelyek WebJobok outputjaként állnak elő a Blob attribútum miatt – ezeket rögtön észreveszi az algoritmus.

# Webalkalmazás létrehozása az Azure Marketplace-ből

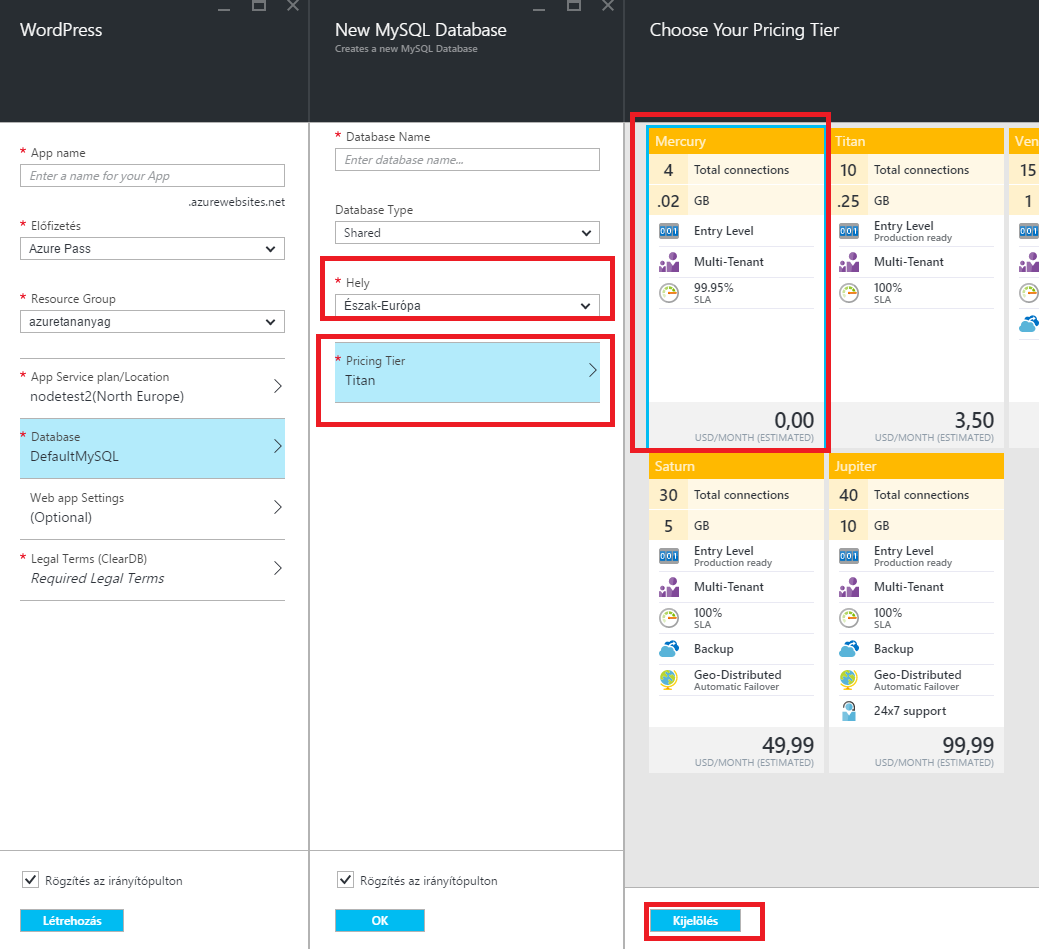
Hozzunk létre egy egyszerű WordPress blogot az Azure MarketPlace-ből! Kattintsunk az Új gombra, majd válasszuk a Piactér felirat mellett az összes megjelenítése gombot:



Ezután ha időnk engedi, kalandozhatunk kicsit a piactéren, vagy akár saját alkalmazásokat is kipróbálhatunk. A gyakorlat során egy egyszerű WordPress site-ot fogunk felhúzni. A keresőbe írjuk be, hogy „WordPress”, majd válasszuk azt a találatot, ahol a gyártó is „WordPress”:



Ezután kattintsunk a létrehozás gombra, majd körülbelül ugyanazon a varázslón kell végigmennünk, mint a webalkalmazás létrehozásakor. Különbség, hogy meg kell adnunk, milyen adatbázist használjon a WordPress Site-unk. A WordPress a MySql adatbázist szereti, így olyat tudunk létrehozni hozzá. A Database alatt érdemes a Merkúr (Mercury) opciót választani, a hely pedig legyen Észak-Európa (az adatbázis neve tetszőleges, de egyedi):



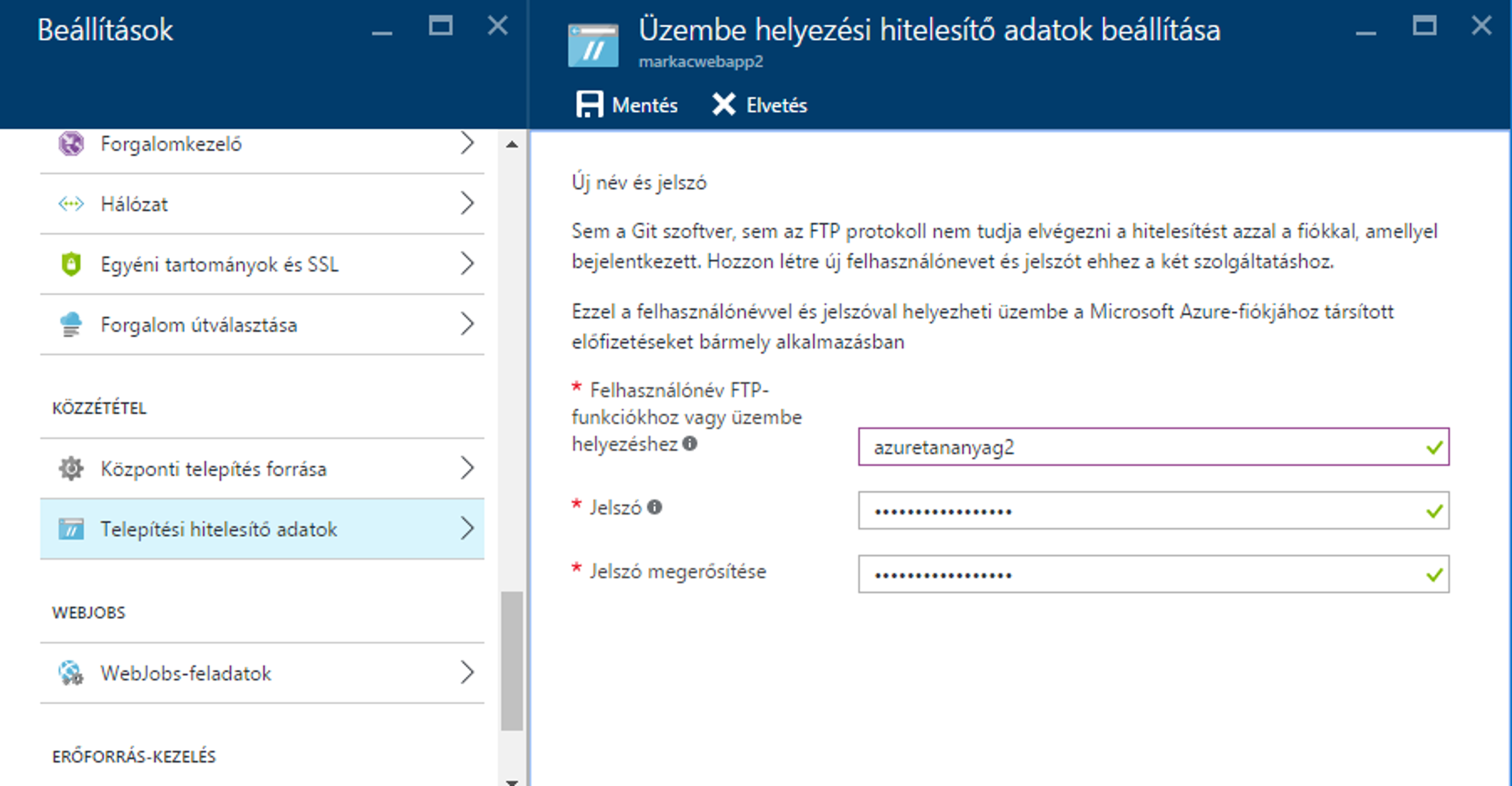
A WordPress adatbázist a ClearDB szolgáltatótól kapjuk valójában, ezért a Required Legal Terms fülre is rá kell kattintanunk egyszer, majd az előugró panelen az OK gombot megnyomva tudomásul vehetjük a szerződés feltételeit. Végül a létrehozás gombbal elkészíthetjük az alkalmazást. Ha létrejött a web app, akkor nézzük is meg. Ilyen egyszerűen készíthetünk például WordPress alkalmazást a Piactérről, de sok egyéb típusú szolgáltatáshoz vannak kész sablonok.

# NodeJS alkalmazás telepítése, Continuous Integration

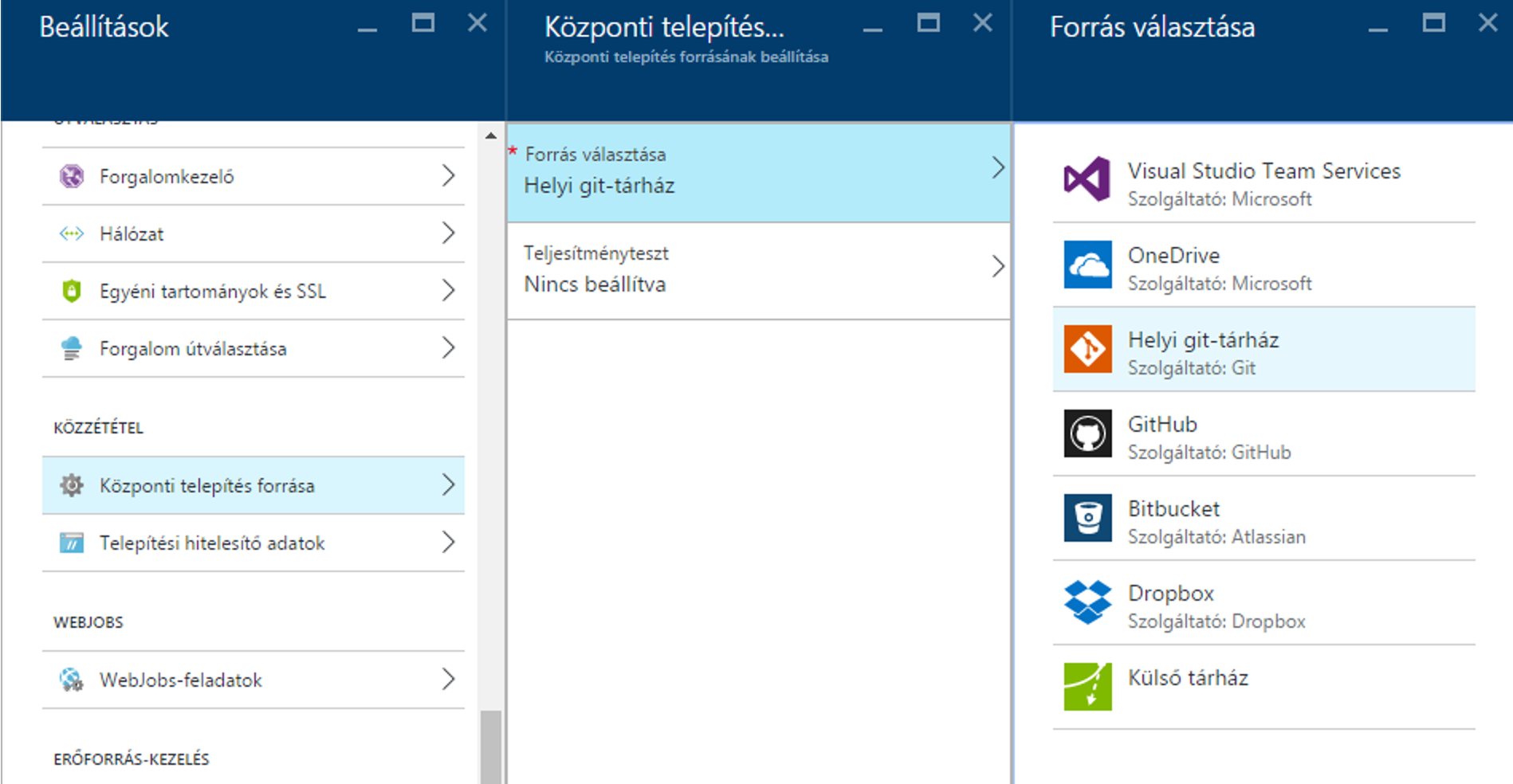
Ebben a részben egy egyszerű NodeJS alkalmazást fogunk elkészíteni és közben bemutatjuk, hogyan tudjuk continuous integration segítségével ezt telepíteni. Fontos, hogy ennek a fejezetnek nem célja a NodeJS ismertetése, a fókusz azon van, hogy egyrészt ezt a platformot is elbírja a web app, másrészt pedig hogyan kell a continuous integrationt beállítani.

## Web App létrehozása, Continuous integration engedélyezése

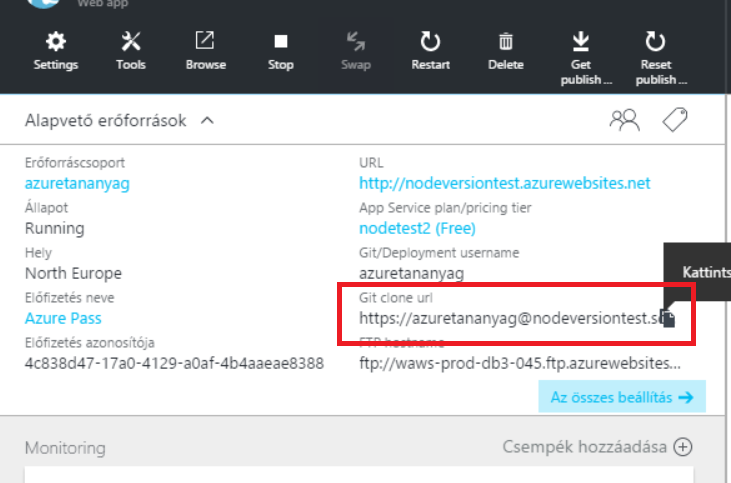
1. Hozzunk létre az Azure Portalon egy új Web App-ot!
2. Mivel a későbbiekben Gitet fogunk használni, ezért a beállítások között először meg kell adni egy felhasználónevet és jelszót, amelyekkel majd a Git felé authentikáljuk magunkat. Ezek tetszőlegesek lehetnek (jelszó legalább 8 karakter, kis/nagybetű ill. szám csoportokból legalább kettőből tartalmaz karaktert), de mindenképpen jegyezzük meg őket:



1. Ezután a Deployment Source alatt be tudjuk állítani, hogy az Azure milyen forrásból húzza be a kódunk legújabb verzióját mindig (lényegében itt állítjuk be a continuous integrationt). Válasszuk ki a Local Git Repository opciót:



Ezzel lényegében létrehoztunk egy Git repository-t Azure-ben a web appunk mögé, ahová kódot tudunk feltölteni. A repository url-jét a web app Dashboardján érjük el. Ezt másoljuk ki a valahova, később szükségünk lesz rá:



1. Következő lépésként a NodeJS alkalmazásunkat fogjuk lefejleszteni. Hozzunk létre valahol a merevlemezen egy üres mappát, ide fogunk dolgozni.
2. Indítsuk el a Visual Studio Code-ot! Válasszuk a File / Open Folder opciót, majd nyissuk meg a mappát, amelyet az előbb létrehoztunk! (Vagy jobb klikk az új mappára és „Open with Code”)
3. Vegyünk fel egy új fájlt (Ctrl+N), majd töltsük fel az alábbi tartalommal:

var http = require('http')

var port = process.env.PORT || 1337;

http.createServer(function(req, res) {

res.writeHead(200, { 'Content-Type': 'text/plain' });

res.end('Hello Node\n');

}).listen(port);

Ez a kódrészlet egyszerűen a „Hello Node” statikus szöveget adja vissza minden kérésre.

1. Mentsük a fájlunkat (Ctrl+S) server.js néven!
2. Ezután indítsuk el a parancssort (Ctrl+Shift+C), majd adjuk ki a következő utasításokat:

git init –Ezzel inicializáljuk a mappában a helyi git repository-nkat

git add . – Minden fájlt hozzáadunk a repository-hoz

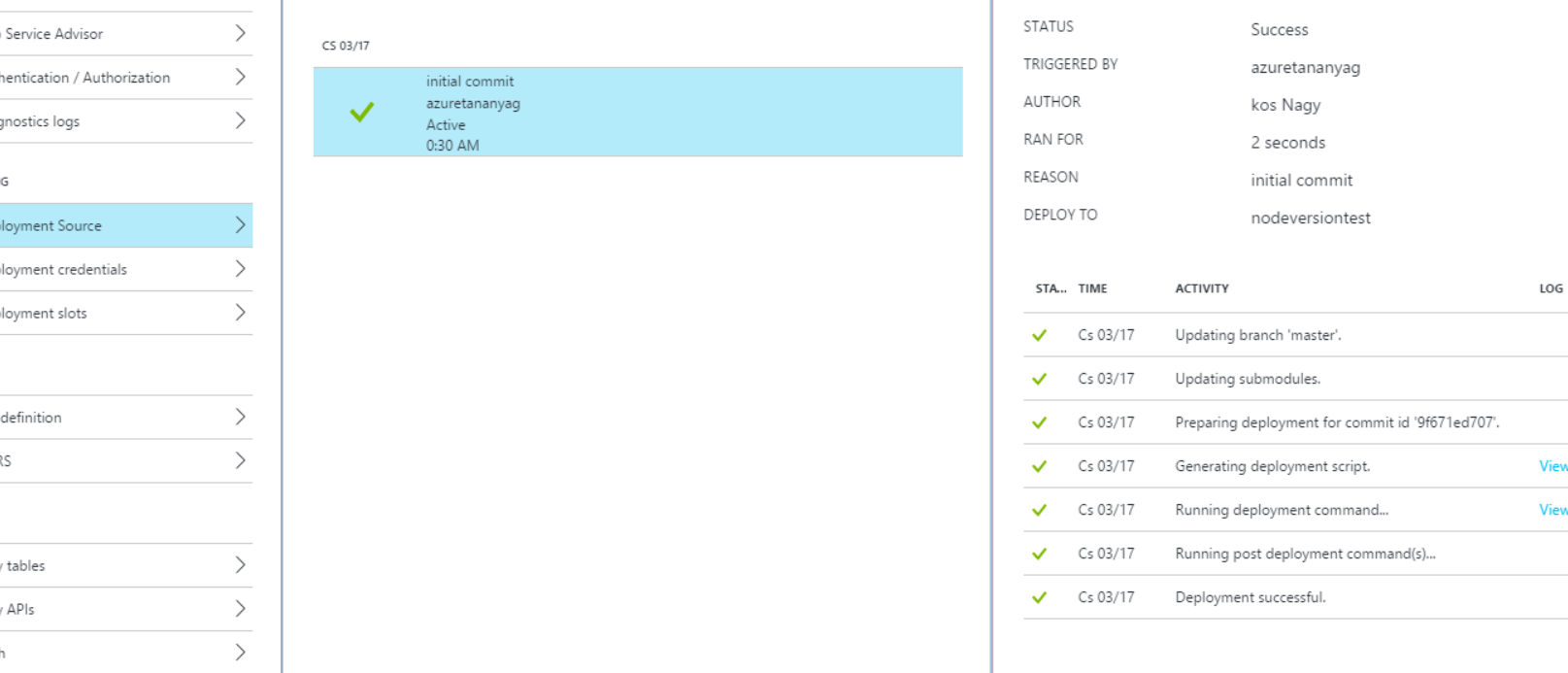
git commit –m “initial commit” – Mentés a lokális repository-ba

git remote add azure https://kimasoltgiturl.git –Felvesszük távoli git repository-ként azure néven a web app repositoryját

git push azure master –Feltöltés a távoli repositoryba

Feltöltéskor a git kéri majd a korábban megadott felhasználónevet/jelszót, ezeket adjuk meg. Ha mindent jól csináltunk, akkor a módosítások feltöltődnek a távoli repository-ba.

1. Ezután a Központi telepítési forrás alatt meg tudjuk nézni, hogy most melyik commitunk van éppen kint élesben, a kitelepítés lépéseit, részletes logokat, de akár újra deployolhatunk egy korábbi verziót is a deploymentek context menüjéből:



1. Végül ha ráhívunk a webalkalmazásra, akkor a Hello Node szöveget fogja megjeleníteni a böngésző (érdemes használni a Dashboard Browse gombját!).

Itt hangsúlyozzuk újra egyrészt, hogy tetszőleges NodeJS alkalmazást ki tudunk például így telepíteni. Másrészt mondjuk el, hogy a continuous integration nem csak NodeJS-sel működik, használhatjuk bármelyik más, támogatott platformot. Harmadrészt pedig ahogy láttuk, nem csak Git, hanem egyéb források is felvehetőek.

# KuduScript

Ebben a feladatban azt fogjuk megnézni, hogyan készíthetünk testre szabott build scriptet, amely segítségével mi magunk irányíthatjuk, hogy mi fut le deployment közben.

1. Indítsuk el az Azure Command Line Interface-t, majd navigáljunk bele a korábban létrehozott NodeJS projekt mappájába!
2. Adjuk ki a következő parancsot:

azure site deploymentscript --node

A –-node kapcsolóval azt adjuk meg, hogy a projektünk típusa egy NodeJS app (itt kellene mást megadni, ha például egy ASP.NET alkalmazáshoz szeretnénk telepítőscriptet). Ha mindent jól csináltunk, akkor megjelenik két új fájl, a .deployment és a deploy.cmd a mappában.

1. A .deployment fájl egyszerűen csak továbbhív a deploy.cmd fájlba, így annak a tartalmát érdemes megnézni. Ebben a fájlban lényegében egy olyan script került most kigenerálásra, amely a korábban is használt, „alapértelmelmezett” continuous integration publikálási folyamatot írja le.
2. Módosítsunk bele picit a deploy.cmd fájlba. Keressük meg az „echo Handling node.js deployment” sort, majd ez alá írjunk bele egy echo Hello Hello Node script sort. Mentsük el a fájlt.
3. Adjuk ki a legutóbbi git parancsok közül a relevánsakat:

git add .

git commit –m “deployment”

git push azure master

Pusholás közben pedig figyeljük a kimenetet; ha mindent jól csináltunk, akkor megjelenik a saját kis echo szövegünk.

Fontos rámutatni, hogy a sorok előtt mindenhol szerepel a „remote” szó, ez azt jelenti, hogy az összes lépés már a távoli gépen, a continuous integration folyamat részeként megy végbe, nem pedig a saját gépünkön! Ez azt jelenti, hogy ha például ASP.NET Web Application lenne a projekt típus, akkor a fordítás (mert azt fordítani is kell) szintén már egy távoli gépen történne.

Ez most persze csak egy kis egyszerű példa volt, de ide bármit beírhatunk – pl.: unit tesztek futtatását.

1. Ha valaki ASP.NET Web Applicationnel is szeretne játszani, annak a paraméterezése kicsit más. Ehhez kell még a project állomány és a „-s” kapcsolóval a solution állomány elérési útvonala (és természetesen a projekt típusának kapcsolója is más):

azure site deploymentscript --aspWAP HelloAzureWebApp\HelloAzureWebApp.csproj –s HelloAzureWebApp.sln